Automated Few-Shot Prompt Generation For and From Large Language Models

Anonymous ACL submission

Abstract

Few-shot prompts are difficult for humans to construct, but can be critical for the performance of large language models on downstream tasks. I propose a framework of automatically generating few-shot prompts by selecting high-quality outputs sampled from the model itself. I apply it to code generation. In testing the framework, I use High Probability Branching, a novel tree-based systematic search, demonstrated to outperform conventional sampling in accuracy and efficiency. I evaluate the performance of the framework by applying it to the GPT-J model with a subset of the HumanEval dataset. The prompt generated by the framework achieves a ten percent relative improvement over model performance with no prompt; the improvement is six times the improvement from the human prompt.

1 Introduction

Few-shot prompting is a common method of providing context to guide large language models (LLMs) to solve problems. Through proper prompting, language models become capable of in-context learning (Dong et al., 2022), gaining the ability to solve complex, multi-step problems (Nye et al., 2021) (Wei et al., 2023). However, it is difficult for humans to create good few-shot prompts because humans often cannot accurately understand or predict the behavior of language models when solving problems.

Automated generation of discrete and continuous prompts has been studied in (Shin et al., 2020) (Liu et al., 2021) for knowledge retrieval, but these methods do not scale well to generating long few-shot prompts which demonstrate multi-step strategies for solving complex problems. This is because these prompt generation methods do not involve a system with semantic understanding of the problems themselves, and thus are unlikely to implement problem-specific strategies. This problem can be avoided by utilizing the output of LLMs, which may include a semantic understanding of the problem, to generate prompts.

Auto-CoT (Zhang et al., 2023) composes few-shot prompts using zero-shot chain-of-thought model question/output pairs as examples. These prompts are demonstrated to be competitive with human-designed few-shot prompts. However, Auto-CoT does not use a measurement of prompt quality or even example correctness in selecting generated examples. As a result, Auto-CoT depends on the particular behavior of the language model in the chain of thought setting, and may not generalize to less powerful models or to domains for which chain of thought is not directly applicable, such as code generation. This may be improved by actively selecting examples that work better for the specific domain by measuring correctness and prompting performance.

I propose an automated framework to generate few-shot prompts. This framework uses the pretrained large language models themselves to generate few-shot prompt candidates and then identifies and selects effective prompts from these candidates. Candidates are generated by searching the space of token sequences generated by the model for those sequences which lead to correct answers for a subset of input problems. Then, good prompts are identified by validating their effectiveness on the other problems from the input.

In this paper, I apply this framework to code generation. Here, the objective is to search for correct output programs which improve code generation performance when used as prompts. Prompting the model with well-written code by humans does not necessarily help the model solve new problems, because large language models interact with code differently from how humans programmers do. In fact, the best few-shot prompts may be unintuitive to humans, and would thus be difficult for humans to compose. Therefore, automated generation of

few-shot prompts not only can reduce need for human labor, but may also produce better prompts than humans can.

In the rest of the paper, I first briefly formulate the problem of prompt generation, casting it as a tree-search problem. I then introduce a five-step automated prompt generation procedure. Next I describe the experimental methodology used to implement and test the framework as applied to code generation, utilizing High-Probability Branching (HPB) as a systematic tree-search algorithm. Finally, I discuss the experimental results.

2 Problem Formulation

2.1 Prompting

Autoregressive large language models together with a sequence sampling algorithm take a tokenized input query Q, and output a probability distribution over generated sequences A. Suppose that there is a dataset of questions $\{q_i\}$ and there exists some measure for whether a model output A constitutes an accurate answer to a particular question q_i. A prompt is a transformation $q_i \rightarrow Q_i$ such that when Q_i is given to the language model, results in an A_i that more accurately answers q_i . As a simplification, assume input query $Q_i = p || q_i$ is simply the concatenation of a prefix p and the question itself q_i where p is static and do not depend on q_i . For human-designed prompts, p often takes the form of instructions, or few-shot examples. It is not necessary in this context for p to generalize to problems outside the domain of $\{q\}$, and indeed it is difficult to imagine fully general instructions or examples for problem-solving across arbitrary domains.

The prompt generation problem is then to find better performing prompts, that is to optimize

$$p_{opt} = \underset{p \in P}{\operatorname{argmax}} \frac{1}{|\{q_i\}|} \sum_{\{q_i\}} \text{Correctness}_{q_i}(M(p||q_i)).$$

Since the space of all possible prompts P is huge, finding the optimal prompt cannot be efficiently computed and heuristic methods must be used. In human prompt design, humans rely on their own intuitions of large language model behavior to choose p. In this paper I propose systematically sampling from model outputs to obtain a candidate subset of P corresponding to few-shot prompts, and selecting the best prompt within that subset.

2.2 Tree-Based Sampling

Mathematically, language models assign a probability to every single possible output sequence of tokens as a factorized product of token probabilities. The entire space of possible generated sequences can be represented as a tree, where each node represents a generated token and descendants represent all the possible tokens that could follow. Running the language model can compute the descendants of a particular node. So for the current node and its ancestors $x^{(1 \dots i)} = x^{(1)} x^{(2)} \dots x^{(i)}$, the language model outputs a probability distribution over all possible next tokens in the vocabulary V , each constituting a child node $x_k^{(i+1)}$ for the kth most probable next token. We can call this exploring the node. Then we have

$$\sum_{k=1}^{|V|} P(x_k^{(i+1)} | x^{(1 \dots i)}) = 1.$$

Sampling sequences from the large language model can be formulated as a tree search problem, where sampling processes, possibly acting in parallel, compute nodes of the tree to find complete sequences which end in terminal nodes. A terminal node might be one that corresponds to an end-of-sequence token, is at a certain depth, or results in a sequence that satisfies a termination condition.

Later in this work, I introduce High Probability Branching, a novel tree-search algorithm.

2.3 Framework Overview

At a high level, the automated prompt generation framework to produce few-shot prompts consists of the following steps:

1. **Initial Input Data:** As input, take a small subset of sample problems representative of the dataset, along with an associated validator. The validator is able to measure the correctness of solutions to the sample problems.

2. **Generation Phase:** Use the language model to generate candidate solutions for the sample problems using some generative sampling strategy.

3. **Selection Phase:** Validate the generated candidate solutions and select high quality ones for evaluation as prompts according to some heuristics.

4. **Evaluation Phase:** Evaluate the performance of selected candidates as prompts by testing them on the sample problems. Select the highest-performing candidates as the output prompts of the system.

3 Experimental Methodology

3.1 Model, Hardware, and Dataset

The model used is GPT-J with 6B parameters (Wang and Komatsuzaki, 2021), released under the Apache License 2.0. Experiments were run using custom input parallelism code across eight 24GB NVIDIA GeForce 3090 GPUs. About 5,000,000 forward passes were performed in total.

The dataset used is the HumanEval dataset (Chen et al., 2021) proposed by OpenAI and released under the MIT license. HumanEval is a set of 164 Python programming problems. Each problem consists of a Python function stub and docstring specifying the expected behavior of the function in natural language, test cases used to evaluate the correctness of proposed implementations, and a human-written canonical reference implementation. Solutions are evaluated on functional correctness, defined by whether a solution passes all provided test cases corresponding to the problem.

I replicate OpenAI's statistics, finding that GPT-J solves HumanEval problems correctly at a rate of 4% at temperature 1 and 10% at temperature 0.2, averaged over 200 samples per question. GPT-J may not be able to comprehend some problems, or may not have the algorithmic capabilities to generate valid solutions even with many samples. This implies that the HumanEval dataset is difficult overall with respect to the capabilities of GPT-J, with around two thirds of the problems being apparently solvable.

To produce a dataset with difficulty more in line with GPT-J's capabilities, I abridge the dataset by considering only problems where both naive sampling and High Probability Branching (discussed later) manage to discover at least one correct solution. This results in an abridged dataset of 37 problems, from which 8 problems are further removed to form an input set, leaving a 29 problem evaluation abridged dataset.

Figure 1 shows the pass@1 accuracy of GPT-J of each problem in the abridged HumanEval dataset at temperature 1 and 0.2, sorted by $T = 1$ performance. Problems that are not shown have basically zero pass@1 accuracy. Abridging the dataset allows for a much more reasonable range of problem difficulty. OpenAI's optimal temperature selection of 0.2 is shown to result in performance improvements that are often significant but inconsistent across problems. For the remainder of the work, temperature 1 is used because tuning the temperature for specific datasets is outside the scope of the research.

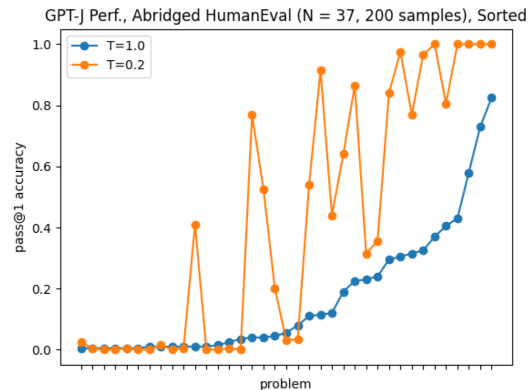

Figure 1: GPT-J pass@1 accuracy, abridged dataset

In order to significantly increase the efficiency of Python code generation, whenever a whitespace token is appended to a sequence more whitespace is added in order to reach the next indentation level if syntactically necessary. Experimentation shows that GPT-J almost never makes errors by emitting an invalid amount of whitespace, so this has minimal effect on generated sequences.

3.2 Top-level Prompt

I introduce the top-level prompt `"Answer Key:\n\n"` before the problem specification and any few-shot prompts. Experimentally, this appears to slightly improve overall performance across all metrics. This top-level prompt may improve instrumental alignment and discourage hallucination and calling functions defined in the prompt, by indicating to the model that the code ought to be correct and that the functions are independent problems rather than components of a larger program.

3.3 High Probability Branching

The best sequence generated from a large language model cannot be solved for directly, and therefore an appropriate sampling algorithm is necessary to produce high-quality sequences. This is true even for the training objective of finding high-probability sequences, but also for alternative

objectives like finding correct programs or good prompts.

I propose a novel, simple tree search algorithm called High Probability Branching (HPB) in order to systematically explore the model output space to find good generated sequences. The parameters for HPB are a token quota Q, which is decremented whenever a node is explored, and an exploration distance L, as well as a termination condition for generated sequences. Define the frontier of the output space to be the set of unexplored children of explored nodes. Initially define to be in the frontier the root node, corresponding to the first token prediction with only the prompt as the input.

HPB consists of repeatedly branching from the highest probability node in the frontier. After identifying the highest-probability frontier node, greedily explore vertically down the tree by repeatedly computing highest-probability children, stopping after a maximum of L children have been explored or if a termination condition for the sequence is reached. Since children are always lower probability than their ancestors, the highest-probability node in the frontier can often be determined before the previous branch ends, allowing many branches to be simultaneously explored in parallel. Branching is suspended when running all active branches to the exploration distance may exceed the token quota. Once the token quota is exhausted, the search ends and all terminated sequences are returned.

HPB can be seen as a generalized beam search. Setting the exploration distance to 1 results in computing the node with highest probability at each step. This guarantees that every terminated sequence is output in order of cumulative probability. If a termination condition of a generation length H is set, this replicates the result of a beam search with horizon H. But similarly to beam search, this becomes infeasible when H becomes larger as the computational requirements grow exponentially. In Table 1 and Table 2, I compare the pass@k accuracy of High Probability Branching with ordinary sampling on the abridged and full HumanEval dataset. The HPB parameters used throughout the work are $Q = 10000$ and $L = 500$. For code generation, sequence termination is determined as soon as the indentation block for the function implementation is broken.

[1]Whether any correct generations were found among all generated sequences for each problem

This evidence suggests that High Probability Branching is superior to naive sampling for code generation in several respects, as it finds more correct solutions at higher accuracy using fewer forward passes. As noted by OpenAI, tuning the temperature to 0.2 significantly increases the pass@1 accuracy on HumanEval, but with the tradeoff of degrading performance on pass@10 and pass@100 (Chen et al., 2021). Highest Probability Branching, in contrast, does not exhibit such tradeoffs as it increases performance against a sampling baseline for the same temperature across all metrics. It is also not mutually exclusive with tuning the model temperature for the specific dataset and metric.

3.4 Evaluation Metrics

I use the pass@k metric for evaluating code generation models (Chen et al., 2021), representing the estimated probability a correct solution is found within k samples. As HPB outputs sequences in a deterministic order, this models the distribution of correct sequences during HPB exploration as uniform.

In addition, I propose pass@kt, the estimated probability a correct solution is found within k forward passes, or k new token generations. Pass@kt better captures the cost-efficiency of using a tree search algorithm to generate completions, while functioning similarly to pass@k for independent sampling algorithms with the added benefit of accounting for different solution lengths between problems.

In Table 3, I compare the pass@kt performance of HPB and ordinary sampling on the abridged dataset. High Probability Branching is able to find twice as many correct generations on average in the same token quota compared to ordinary sampling.

4 Implementation and Results

4.1 Initialization

As an input to the prompt generation framework, I select every fifth problem from the abridged HumanEval dataset, for a total of 8 problems and associated test cases. This splits the 37 problems into an input set and a test set. The 8-problem input set and its test cases are provided to the framework and are used to generate prompts. The 29-problem test set is used afterwards to evaluate the performance of the prompts and judge the efficacy of the automated prompt generation framework.

Table 1: Sampling Strategy Performance, Full Dataset (N = 164)

Generation Strategy	pass@any[1]	pass@1	pass@10	pass@100
HPB, T=1, Answer Key prompt	**0.2988**	0.0639	**0.1605**	**0.2734**
Sampling, T=1, Answer Key prompt	0.2927	0.0427	0.1375	0.2477
Sampling, T=1, No prompt	0.2805	0.0401	0.1270	0.2300
Sampling, T=0.2, No prompt	0.2561	**0.1030**	0.1533	0.2200

Table 2: Sampling Strategy Performance, Abridged Dataset (N = 29)

Generation Strategy	pass@any	pass@1	pass@10	pass@100
HPB, T=1, Answer Key prompt	**1.0000**	0.2703	**0.6813**	**0.9484**
Sampling, T=1, Answer Key prompt	**1.0000**	0.1834	0.5877	0.9329
Sampling, T=1, No prompt	**1.0000**	0.1772	0.5502	0.8921
Sampling, T=0.2, No prompt	0.7586	**0.4272**	0.6096	0.7192

4.2 Candidate Generation

I use High Probability Branching with $Q = 10000$ and $L = 500$ to generate candidate prompts for each of the 8 problems. In this work, candidate prompts will be referred to by a two number code **p-n** for the nth generated sequence from the pth HumanEval problem.

4.3 Candidate Selection

I discard all generated solutions that are not functionally correct by testing them against the known test cases. I select up to 8 possible candidates from the set of correct solutions generated for each problem by simply choosing the first four correct sequences to be produced during the generation process.

In order to select candidates from the problems with more than eight correct generated sequences, I originally considered choosing the ones with the highest mean log probability per token. OpenAI found that mean log probability, but not sum log probability, is modestly associated with functional correctness(Chen et al., 2021). However, selecting based on this criteria results in very semantically similar candidates, which for example differ only in the name of a variable or in the number of line breaks.

Instead choosing the first four correct sequences to be produced during the generation process results in a more diverse set of candidates, because sequences which terminate early also are likely to have branched early.

Table 4, shows each of the eight problems in the input set, how many sequences generated for each, the number of sequences that were correct, and finally the number of candidates selected for evaluation.

4.4 Prompt Evaluation

To evaluate the performance of the candidate prompts, I use them as one-shot prompts and perform HPB to generate solutions to the seven other problems in the known input data. For each of the prompts and problems, I compute five performance metrics: pass@1, pass@10, pass@100t, and pass@1000t. I also treat the canonical solution for problem 58 in HumanEval as a candidate prompt as if it were generated by the model, and evaluate it the same way.

Each prompt has performance metrics on one of the prompts missing, since a prompt cannot be fairly evaluated on the problem it was derived from. Therefore for each problem I use the mean performance metrics over all prompts which could be evaluated on that problem as normalized placeholders for the prompts that were derived from that problem. Prompts are ranked for selecteion by pass@1.

Table 5 shows the pass@k and pass@kt results for the two best and two worst-performing prompts as well the human-written canonical prompt. **58-13**, **58-15**, and **28-1** are the three prompts with highest pass@1. From these **58-13** and **28-1** are selected as the final output of the prompt generation framework, found by searching the output space of eight original sample problems for high-performance prompts.

4.5 Validation of Generated Prompts

In order to validate the performance of the framework, I test **58-13** as a one-shot prompt, **28-1** and **58-13** together as a two-shot prompt, as well as

Table 3: Token Efficiency, Abridged Dataset (N = 37)

Generation Strategy	pass@1	pass@100t	pass@1000t	Correct Samples	Tokens Generated
HPB	**0.2703**	**0.4457**	**0.7958**	**73.6897**	9332.6897
Sampling	0.1834	0.3043	0.6717	36.6897	10329.2069

Table 4: Prompt Generation and Selection

Problem	Tokens Generated	Samples Generated	Correct Samples	Selected Candidates
0	10036	125	3	3
8	10004	191	86	8
18	10029	193	4	4
28	10030	561	405	8
35	10022	254	159	8
51	10034	186	4	4
58	10032	167	94	8
152	10057	166	6	6

the human-written canonical solution to 58 as a one-shot prompt. I also test the worst prompt **152-151** identified by the framework. I compare all of these against the previously evaluated no-prompt baseline. Table 6 shows the pass@k and pass@kt performance of these prompting options on the 29 remaining problems in the abridged dataset, again using HPB with $Q = 10000$ and $L = 500$.

4.6 Results and Discussion

The accuracy data in Table 6 supports the effectiveness of the framework at generating good prompts. Prompts identified by the framework as good through validation on the limited input data are more effective when tested on held-out data. Additionally, the prompt identified as bad by the framework harms performance when used with the held-out data, despite being a correct solution. The framework's selection process is therefore able to measure the intrinsic quality of prompts.

The best one-shot prompt **58-13** improves pass@1 performance over baseline from 27% to 30%, while the human-written canonical solution is much less effective at 27.5%. This represents a ten percent improvement in pass@1 performance, six times that of the human-written prompt.

For pass@10 and pass@100, the benefits of few-shot prompting in code generation are not demonstrated, as the performance of not using a prompt is competitive with or superior to using various prompts. This can be explained by few-shot prompting decreasing the diversity of generation, similarly to reducing the temperature. Additionally, during the selection phase of the experiment, prompts were chosen based on their pass@1 performance, not their pass@10 or pass@100 performance. Pass@10 and pass@100 performance is slightly improved by combining high performing prompts **28-1** and **58-13** at only a marginal pass@1 performance tradeoff. This is explained by the combined prompt recovering some diversity in sequence generation, which indicates a possible benefit of prompting using multiple examples.

Finally, for pass@100t and pass@1000t, the two-shot combined prompt outperforms all other prompts and no prompt. This means that the combined prompt results in the most token-efficient discovery of correct programs on the dataset. This may be because the selected prompts represent strategies that result in shorter programs, meaning fewer forward passes are spent exploring long, incorrect solutions. For many problems, the model attempting a short solution may also more likely result in a correct program than a long solution, because there are less opportunities for the model to make mistakes.

Inspecting the highest-performing prompts derived from problem 58 reveals that the two are closely related. As shown in Figure 2, **58-15** is a "clean" solution that solves the problem in one line with set operations and the `sorted()` function, while **58-13** is a "dirty" solution that is identical to 58-15 except that it wraps the return value in a list comprehension. Since `sorted()` already returns a list, the list comprehension simply iterates through the answer and copies it to a new list, which is useless from an algorithmic standpoint. Most human programmers would probably prefer to remove the

Table 5: Prompt Evaluation: HPB Prompt Performance on Input Set (N=8, selected)

Prompt	pass@1	pass@10	pass@100	pass@100t	pass@1000t
58-13	**0.3578**	**0.7538**	0.9928	**0.4576**	**0.8617**
58-15	0.3557	0.7226	0.9982	0.4435	0.8442
58 canonical	0.3288	0.7084	**0.9988**	0.4169	0.7849
152-84	0.2704	0.7398	0.9943	0.4499	0.8557
152-151	0.2008	0.6548	0.9783	0.4323	0.8385

Table 6: HPB Prompt Performance, Abridged Dataset (N = 29)

Prompt	pass@1	pass@10	pass@100	pass@100t	pass@1000t
No prompt	0.2703	0.6813	**0.9484**	0.4457	0.7958
28-1+58-13 (two-shot)	0.2966	**0.6839**	0.8920	**0.4958**	**0.8101**
58-13	**0.2981**	0.6643	0.8636	0.4928	0.7874
58-15	0.2794	0.6590	0.8670	0.4852	0.7831
58 canonical	0.2755	0.6724	0.8714	0.4656	0.7785
152-151	0.2326	0.5805	0.8098	0.4656	0.7316

unnecessary list comprehension, because it makes the code less efficient and adds clutter. A huma prompt designer might reason that a prompt with an extraneous list comprehension provides a context indicative of a low-skill programmer, and for that reason expect the prompt to underperform the "clean" version due to promoting harmful behaviors.

The prompt generation framework selects the "dirty" solution as the preferred prompt, in defiance of human intuition. In order to judge whether or not this is an erroneous selection brought about by variance or bias in the evaluation process, I validate both 58-13 and 58-15 as prompts against the 29-problem test set. Indeed, the dirty prompt continues to significantly outperform the clean prompt on the larger test set, which is strong evidence the selection was not erroneous and the dirty solution is truly a better prompt for code generation with GPT-J.

One possible explanation for this counterintuitive result it might be a good strategy to begin with a list comprehension when a output list is expected, even if it is inapplicable in this particular case. The branching point between the two prompts occurs directly after the `" return"` token, where the clean solution emits `" sorted"` and the dirty solution emits `" ["`. Emitting `" sorted"` is correct because the `sorted()` function returns a list and can be used to solve the problem. Emitting `" ["` on the other hand guarantees the output will eventually be a list, which can be seen as a small step towards solving the problem. Entering a list comprehension environment when trying to return a list is at worst harmless, and allows for useful strategies such as converting a non-list iterable to a list. Therefore, promoting more use of list comprehensions, even when unnecessary, can be seen as encouraging a conservative strategy for solving list manipulation problems by taking a small step while leaving options open. This strategy might improve the likelihood of correctness in the general case, while possibly being difficult for humans to think of.

4.7 Time Complexity and Effects of Input Size

The time complexity of the prompt generation procedure is $O(KQN^2)$, where K is the number of prompts per input problem, Q is the token quota when measuring prompt accuracy during selection, and N is the number of input problems. Increasing K widens the candidate pool, making it more likely a good prompt will be proposed, while increasing Q increases the precision of the prompt quality estimate during selection, making it more likely the best prompt will actually be selected. However, it is less clear what effect N has, since it characterizes both how well the input set represents the problem domain, as well as the number of possible starting points for prompts.

In order to investigate the effect of the input set size N on the generated prompt, I perform an input ablation analysis by considering what prompt would have been chosen as the best prompt during the selection phase for smaller values of N. To do this, I run the selection phase on all possible

```python
def common(l1: list, l2: list):
    """Return sorted unique common elements for two
    ↪  lists.
    >>> common([1, 4, 3, 34, 653, 2, 5], [5, 7, 1,
    ↪  5, 9, 653, 121])
    [1, 5, 653]
    >>> common([5, 3, 2, 8], [3, 2])
    [2, 3]

    """
    return [i for i in sorted(set(l1) & set(l2))]
```

```python
def common(l1: list, l2: list):
    """Return sorted unique common elements for two
    ↪  lists.
    >>> common([1, 4, 3, 34, 653, 2, 5], [5, 7, 1,
    ↪  5, 9, 653, 121])
    [1, 5, 653]
    >>> common([5, 3, 2, 8], [3, 2])
    [2, 3]

    """
    return sorted(set(l1) & set(l2))
```

Figure 2: Prompt **58-13** (left) and Prompt **58-15** (right).

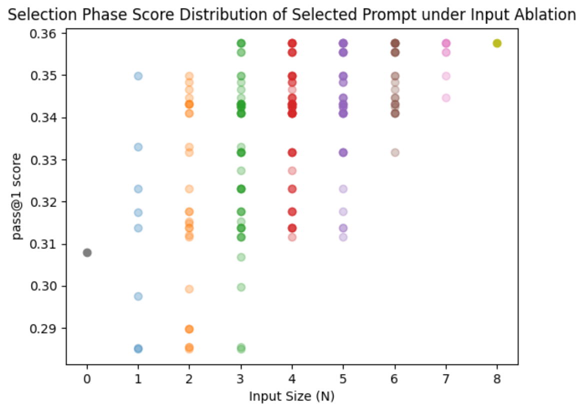

Figure 3: Distribution of estimated performance of selected prompt with ablated inputs

subsets of the input set, where each subset of a fixed N is considered equally likely. It is then possible to observe the rate at which each prompt is ranked as the best one for each value of N. In Figure 3, the data is plotted to show the range of selected prompt quality for each value of N, using the pass@1 quality estimation from the $N = 8$ selection phase, including the corresponding figure when no prompt is used ($N = 0$).

The ablation data supports the coherency of the prompt selection process. As N increases, the probability of selecting the best prompt **58-13** steadily increases, as does the probability of selecting the second-best prompt. However, this also means the quality of prompt selection may depend strongly on N. Even reducing the input set by just one problem to 7, there is a 1/4 chance of erroneously selecting **58-13** over **58-15**, despite the robust 2% performance gap measured on the 29-problem dataset. An input set size of three or less incurs the risk of selecting a prompt which harms performance compared to no prompt. When N is lower the chance of problem 58 not being included at all increases, meaning the selected prompt must be derived from a problem less likely to produce good prompts. This implies that better prompts may yet still be found if the input size were larger than 8, which encourages future improvements in scaling prompt selection to higher input sizes if input data is available.

If it was assumed more known data was available, it would be beneficial to efficiently use that data to find more optimal prompts, but doing so naively may become infeasible at large N. One possible improvement may be to dynamically adjust Q during prompt evaluation in order to allocate more computational time to measurements that provide more information. For example, the system could track as a prior the intrinsic difficulty of a problem based on achieved performance, and if a problem is observed to be so difficult all prompts tend to have near-zero performance, computation could be reallocated to problems with more differentiation in prompting performance instead.

5 Conclusion

This paper proposes an automated prompt generation framework to automatically produce few shot prompts for large language models by sampling, evaluating, and selecting outputs from the models themselves. A novel tree-based algorithm, High Probability Branching, is devised to increase efficiency and accuracy of sampling candidate prompts from the models. The framework is tested by applying it to create prompts for python code generation. The prompts automatically produced by the framework are found to produce a ten percent performance improvement in generating correct Python code solutions to programming problems in the HumanEval problem set. Furthermore, generated prompts perform significantly better than the human-written solution used as a prompt.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. Gpt understands, too. *ArXiv*, abs/2103.10385.

Maxwell Nye, Anders Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *ArXiv*, abs/2112.00114.

Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Eliciting knowledge from language models using automatically generated prompts. In *Conference on Empirical Methods in Natural Language Processing*.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2023. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*.

A Limitations

This study has some limitations. First, the scope of the experiments were limited. The model GPT-J was used with the dataset HumanEval. GPT-J is a relatively small model. Additionally, the dataset was abridged to match the capabilities of the model. Moreover, the result was obtained using the particular HBP parameters $Q = 10000$ and $L = 500$. Therefore, the experimental results should be considered preliminary. Further experimentation with larger models, unabridged datasets, and other domains is desirable.

Second, the measured prompt performance in this study may be different from the benefit for a user using generated prompts. The metrics measured were pass@k and a similar metric pass@kt using the standard estimator proposed in (Chen et al., 2021) and HPB with parameters $Q = 10000$ and $L = 500$ as a sampling method. However, the result obtained by a user depends not only on the prompts but also on how the prompts are used. A user of a prompt may generate only a few samples using naive sampling or a lower token quota with HPB, obtaining different results.

B Risks

Large language models may be misaligned to human intentions, so there is a risk of producing biased, incorrect, or dangerous outputs, and in the context of code generation they may produce biased, incorrect, or dangerous code. Because the proposed automated prompt generation procedure uses model outputs to generate prompts, generated prompts and outputs derived from them are also subject to these risks. At present, human review of model-generated code is necessary before use in real applications.